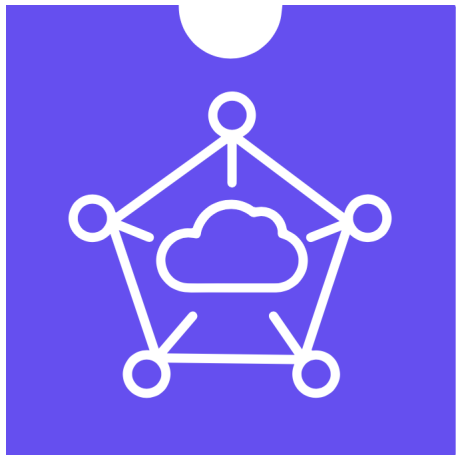
Scuola di Ingegneria e Architettura
Dipartimento di Informatica - Scienza e Ingegneria
Corso di Laurea in Ingegneria Informatica

Supporti Runtime ad Alte Prestazioni per WebAssembly: il Caso di WasmEdge



Relatore:
Prof. Paolo Bellavista

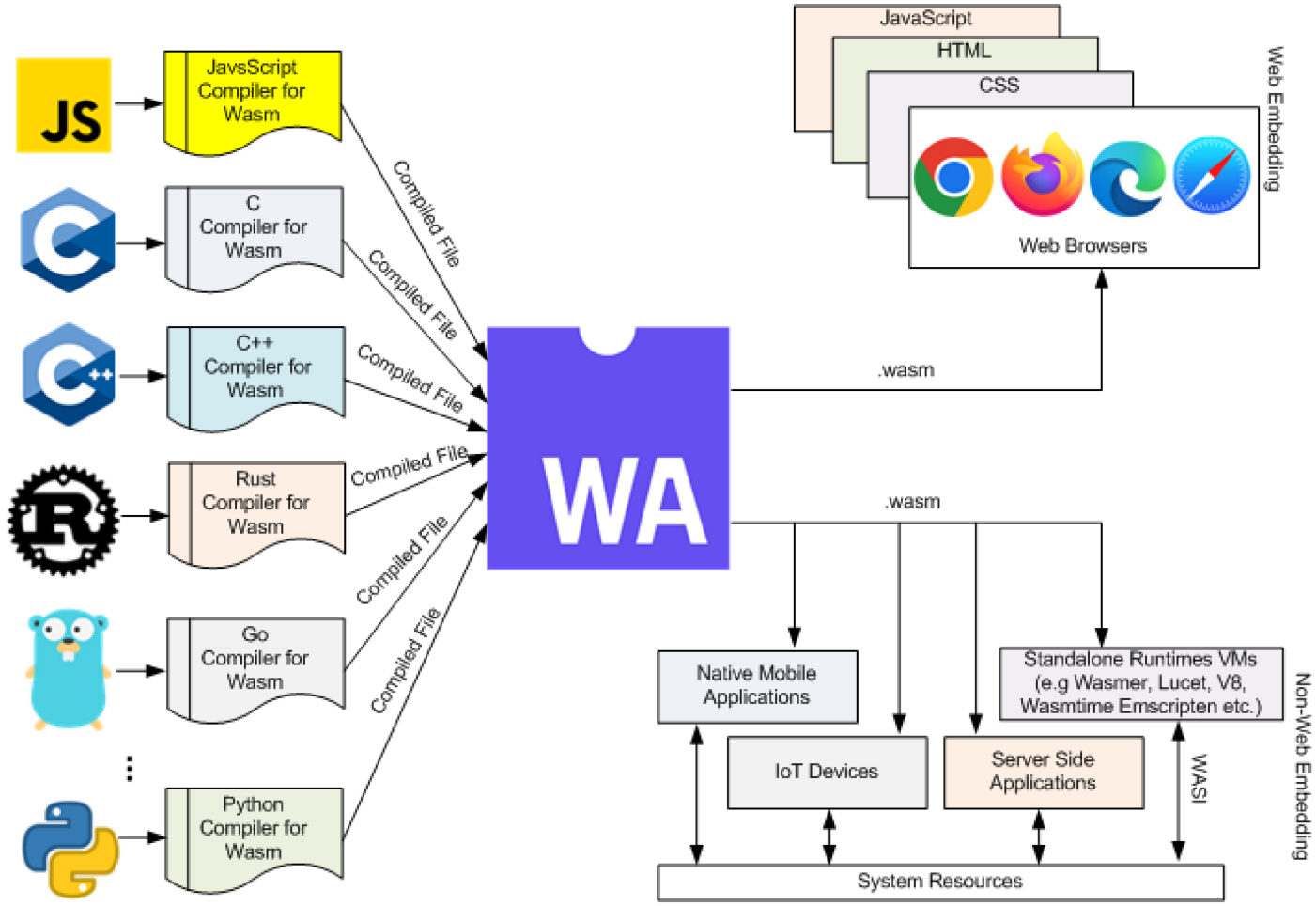
Presentata da:
Lorenzo Pellegrino

Anno Accademico 2022/2023

Obiettivi della tesi

- Studiare le fondamenta tecnologiche di WasmEdge
- Studiare il suo funzionamento
- Studiare il contesto non-browser in cui si inserisce
- Realizzare un prototipo per stressarne le capacità e capirne le potenzialità

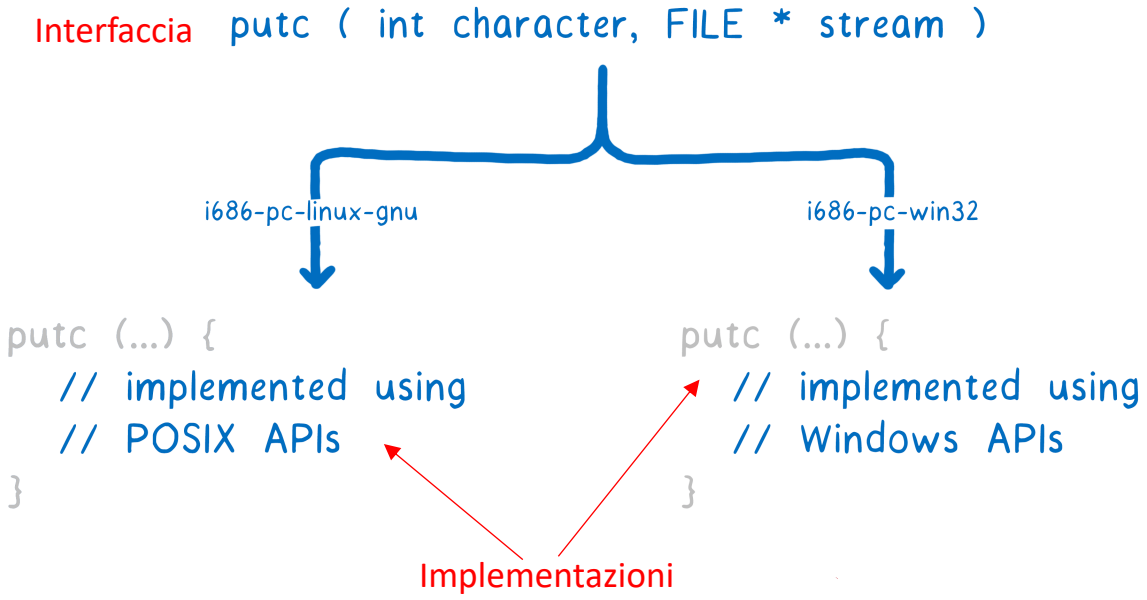
La tecnologia: WASM

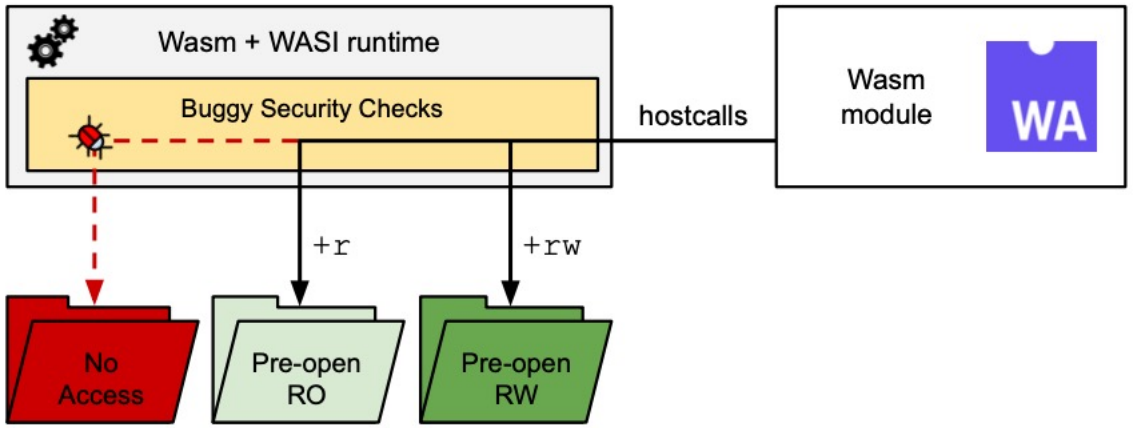


Wasm (WebAssembly):

- È un formato bytecode (.wasm)
- Fornisce performance eccellenti, creato per supportare JS nel browser
- Supportato da vari linguaggi (es. Rust)
- Eseguibile su qualsiasi runtime Wasm, indipendentemente dall'architettura
- Esecuzione sicura con sandbox

La tecnologia: WASI





WASI (WebAssembly System Interface):

- Ridefinisce le **chiamate di sistema** (es. open, read,…) del sistema sottostante tramite delle interfacce, implementate poi secondo l'architettura sottostante
- La ridefinizione delle system-call aumenta la **compatibilità** dell'applicazione e facilità la distribuzione e lo svuluppatore
- All'interno delle interfacce sono presenti anche dei controlli per garantire l'accesso sicuro al filesystem tramite le **capability**, assegnate dal runtime all'avvio
- L'accesso limitato alle risorse rende ancora più forte la sandbox di esecuzione

Il Funzionamento

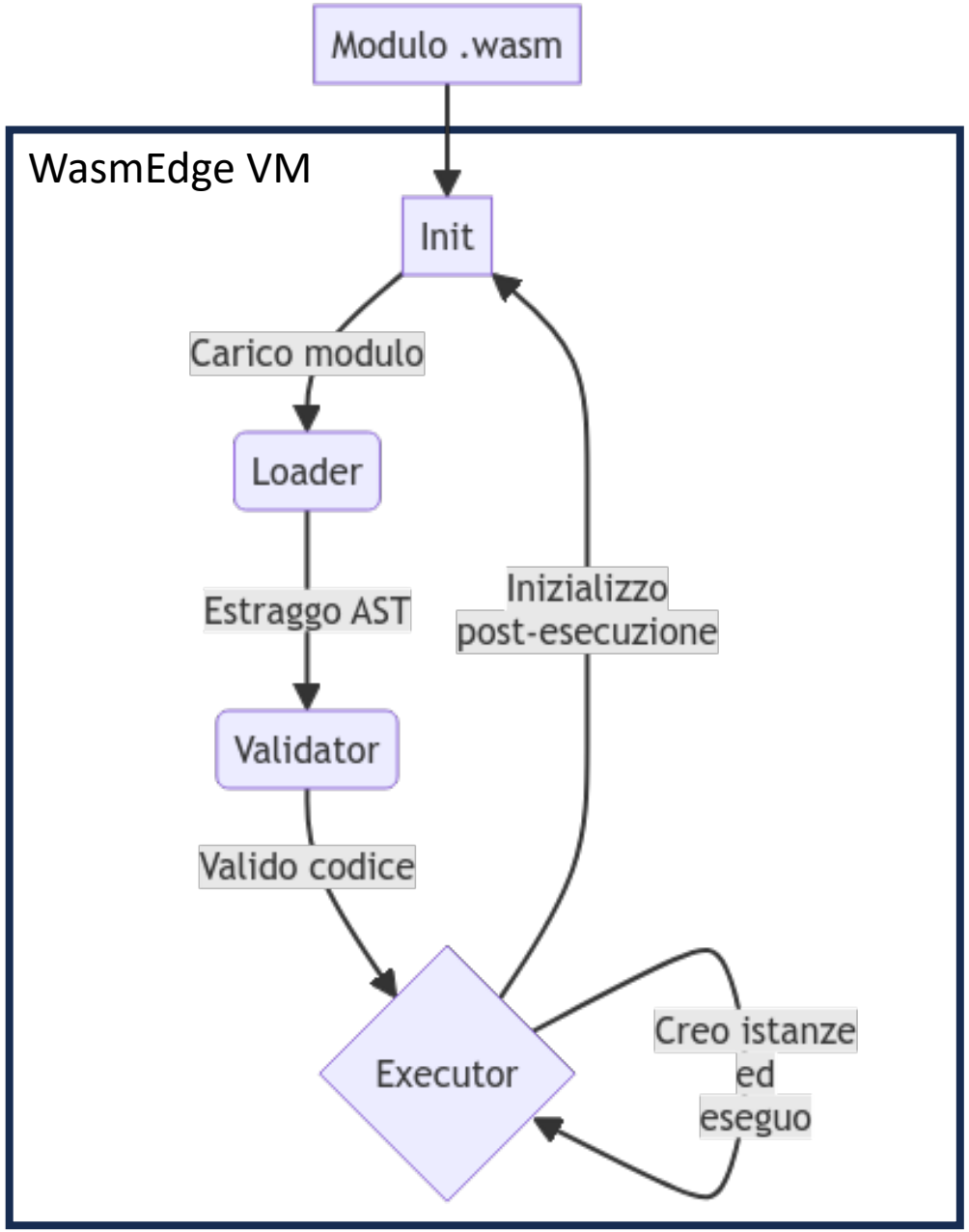
L'esecuzione avviene nella VM interna secondo 4 fasi:

1. **Inizializzazione** (Init)

2. **Caricamento** (Loader)
   - Estrae albero sintattico
   - Individua eventuale sezione AOT

3. **Validazione** (Validator)

4. **Esecuzione** (Executor)
   - Vengono create le istanze richieste nell'AST
   - I **manager della memoria** implementano la sandbox durante l'esecuzione
   - Le **interfacce per le funzioni esterne** permettono l'uso di WASI, funzioni standardizzate e plug-in personalizzati, durante l'esecuzione



Prototipo: struttura

Il SW è stato realizzato in **Rust** (WasmEdge, Wasmtime, Wasmer) e in **JavaScript** (Node.js)

L'applicazione si basa su 3 fasi:

1. Lettura di un file con dimensione a scelta tra 7 ordini di grandezza (da 1KB a 500MB)
2. Ordinamento del contenuto (BubbleSort, MergeSort, QuickSort)
3. Scrittura del contenuto

Le esecuzioni sono state automatizzate con **auto.sh** e le misurazioni sono state effettuate tramite **benchmark.sh** (CPU, Memoria e Latenza)

Sono state effettuate medie contando **15 esecuzioni** per ogni combinazione e per ogni runtime, evitando le combinazioni impossibili

```
PROTOTIPO
├── auto.sh
├── benchmark.sh
├── combinazioni_impossibili.txt
├── documentazione.md
├── JS_version
├── misurazioni.txt
├── RUST_version
└── Rilevazioni_runtimes.xlsx
```

Prototipo: Utilizzo della memoria

Le esecuzioni in base all'algoritmo di ordinamento, hanno mostrato un andamento prevedibile sia in termini di uso della memoria e sia in termini di latenze. Considerando però solo il MergeSort ad esempio:

Carichi	**Node.js**	**WasmEdge**	**Wasmtime**	**Wasmer**
Leggeri (1KB, 10KB, 100KB)	18 - 26 MB	1 - 15MB	1 - 3MB	0.5 - 3MB
Medi (1MB, 10MB)	X	33 - 103MB	7 - 80MB	32 - 105MB
Alti (100MB, 500MB)	X	0.8 - 3GB	0.7 - 3GB	0.8 - 3GB

Osservazioni:

- Dopo i carichi leggeri Node.js è risultato eccessivamente inefficiente e non è stato misurato
- WasmEdge non spicca tra i suoi competitor
- All'aumentare del carico corrisponde una minore disparità tra i runtime

Prototipo: Latenza

Le esecuzioni in base all'algoritmo di ordinamento, hanno mostrato un andamento prevedibile sia in termini di uso della memoria e sia in termini di latenze. Considerando però solo il MergeSort ad esempio:

Carichi	**Node.js**	**WasmEdge**	**Wasmtime**	**Wasmer**
Leggeri (1KB, 10KB, 100KB)	202 - 236ms	73 - 200ms	33 - 51ms	80 - 118ms
Medi (1MB, 10MB)	X	~ 20s	0.2 - 2s	0.6 - 3s
Alti (100MB, 500MB)	X	22 - 108s	20 - 105s	30 - 230s

Osservazioni:

- Node.js abbastanza più lento dei runtime Wasm
- WasmEdge decisamente più lento dei suoi competitor, soprattutto nei carichi medi e piccoli
- WasmEdge diventa più competitivo nei carichi alti
- Wasmtime si può considerare il più efficiente e veloce

Conclusioni

Stato attuale

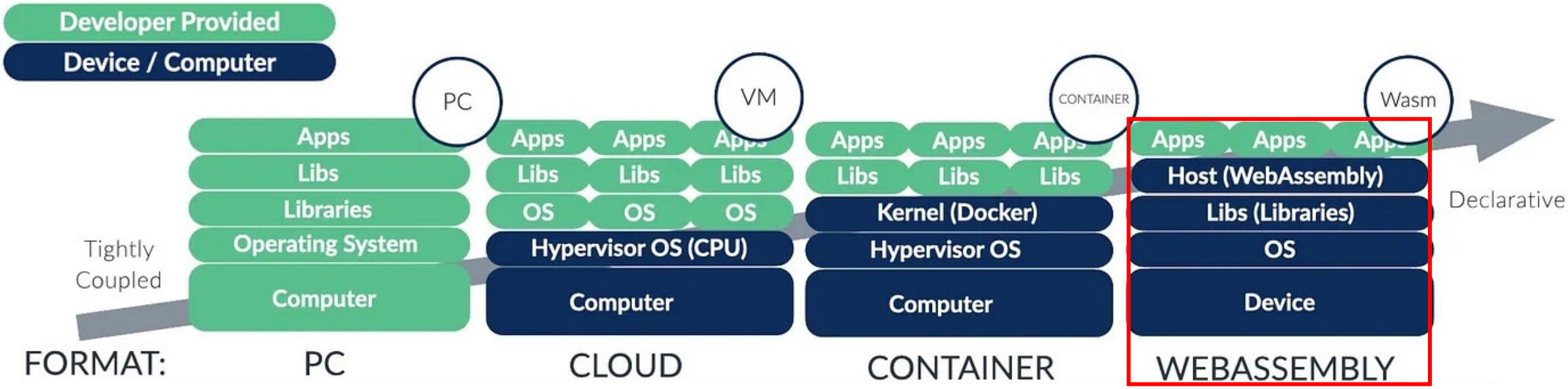
- WasmEdge è un'alternativa efficiente e veloce a Node.js
- Mostra risultati generalmente inferiori rispetto ad alcuni runtime Wasm concorrenti
- Competitivo con i carichi alti

Sviluppi futuri

- Si devono standardizzare ancora molte proposte tra cui il multithreading in WASI (proposta in fase 1)
- Utilizzabile per l'implementazione e il testing di proposte ancora non standardizzate tramite le estensioni

Grazie per l’attenzione

Le fondamenta - 1 : Architetture

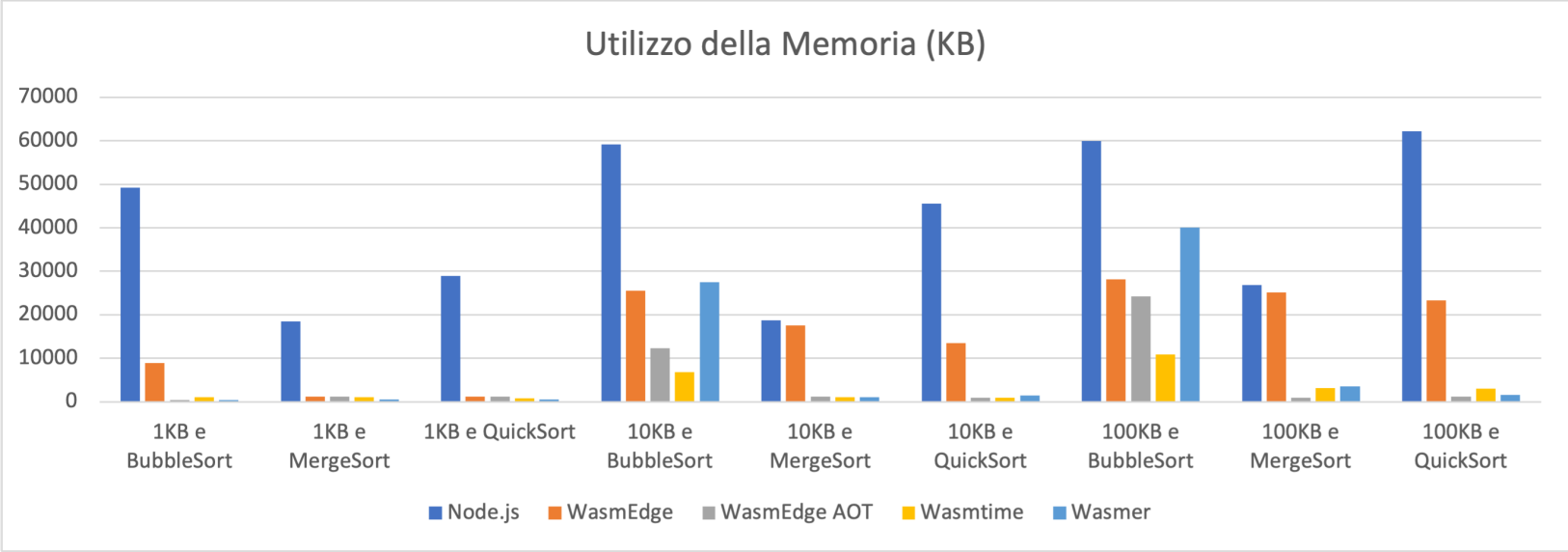


La distribuzione richiede:

- Isolamento dai livelli sottostanti
- Sicurezza delle applicazioni
- Maggiore leggerezza delle applicazioni
- Maggiore disponibilità dei servizi →
- Velocità
- Scalabilità
- Efficienza

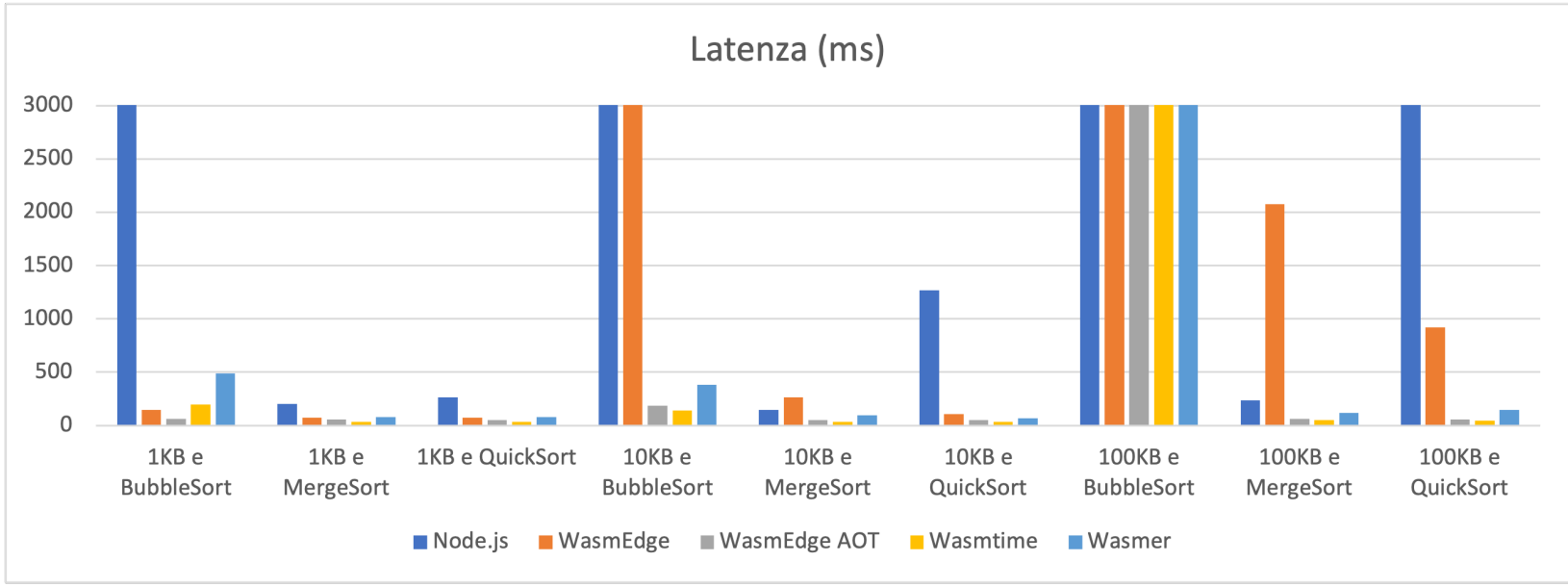
disponibilità %	downtime per anno	downtime per mese	downtime per settimana
98%	7,3 giorni	14,4 ore	3,36 ore
99%	3,65 giorni	7,20 ore	1,68 ore
99,5%	1,83 giorni	3,60 ore	50,4 minuti
99,9%	8,76 ore	43,2 minuti	10,1 minuti
99,99%	52,6 minuti	4,32 minuti	1,01 minuti
99,999%	**5,26 minuti**	**25,9 secondi**	**6,05 secondi**
99,9999%	31,5 secondi	2,59 secondi	0,605 secondi

Prototipo: Node.js vs Runtime Wasm



Utilizzo della Memoria:

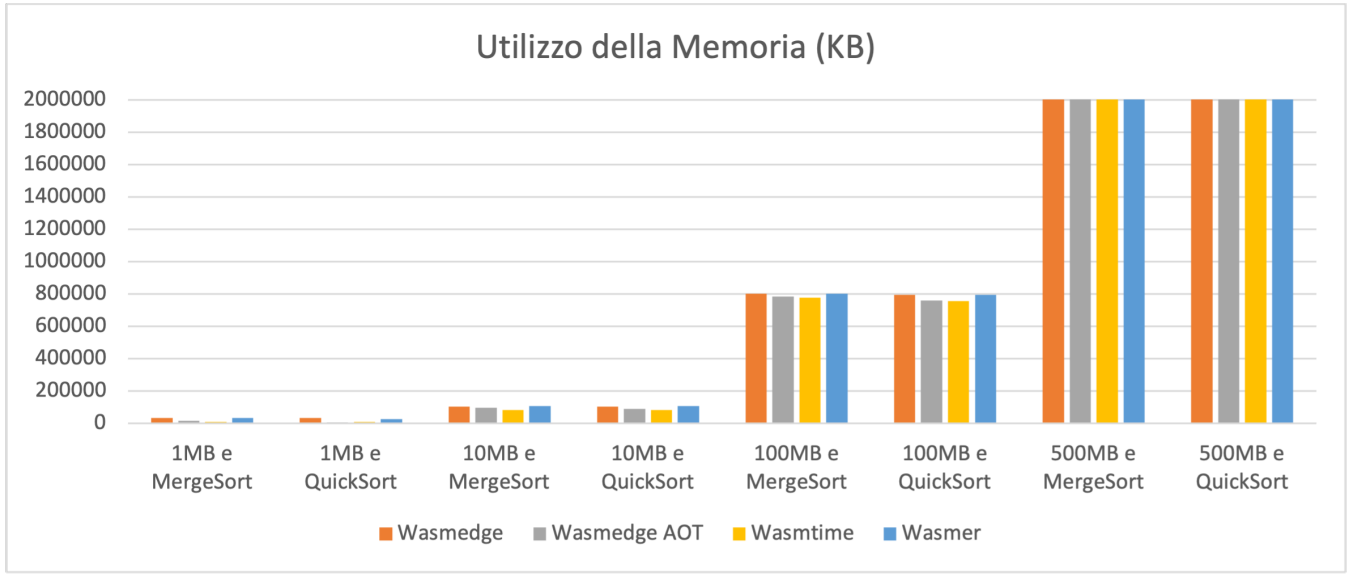
- Differenza in ordini di grandezza
- WasmEdge nettamente peggiore dei competitor



Latenza:

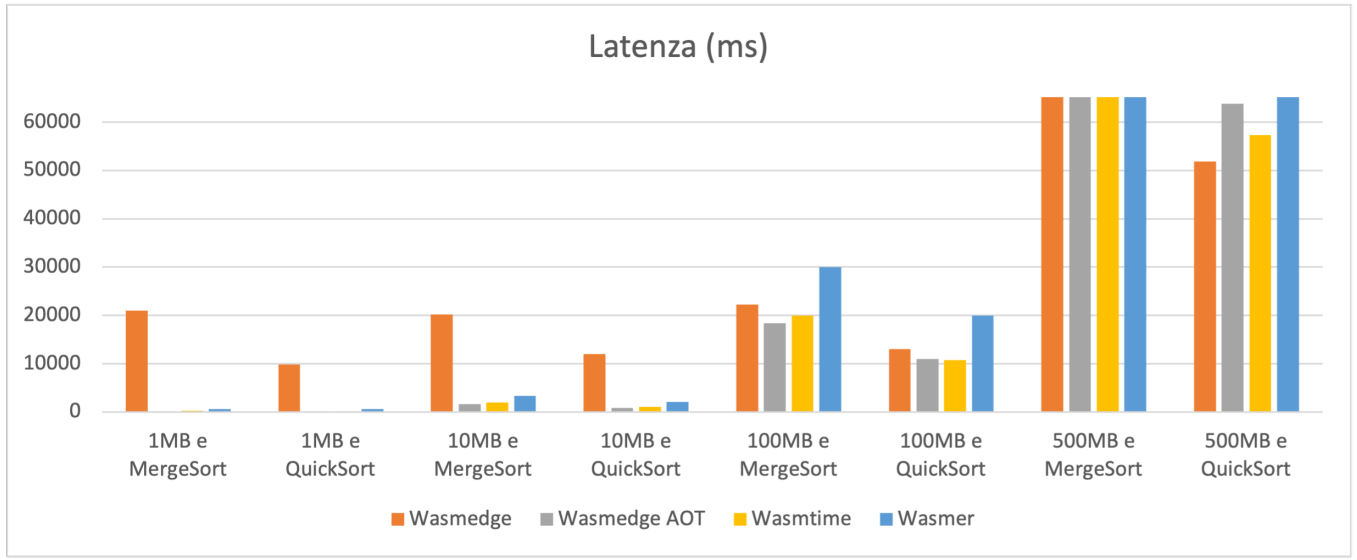
- Differenza di qualche ordine di grandezza
- WasmEdge presenta dei valori più alti rispetto ai suoi competitor

Prototipo: WasmEdge vs Wasmtime vs Wasmer



Utilizzo della Memoria:

- Differenze principalmente dovute al carico, quindi generalmente costante in funzione dell'input
- WasmEdge si dimostra più competitivo con i grandi carichi rispetto ai medio/bassi



Latenza:

- WasmEdge decisamente più lento, con differenze anche nell'ordine delle decine di secondi
- WasmEdge diventa più veloce rispetto ai competitor con carichi medi